1. 上课约定须知

课程主题：Flink 源码解析 -- 第四次课（JobMaster 启动，联系 ResourceManager 申请 Slot，然后部署 Task 执行）
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

第三次：Flink Job 的提交（Client的提交 和 四层图架构）

2. 上次课程作业

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

```
1、资源集群主节点叫做： ResourceManager，负责管理整个集群的资源
2、资源集群从节点叫做： TaskExecutor，负责提供资源
3、ResourceManager 启动的时候，要启动一个验活服务，制定一种机制（比如：某个 TaskExecutor 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
4、TaskExecutor 启动之后，需要向 ResourceManager 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 ResourceManager 之间的心跳以便 ResourceManager 识别到 TaskExecutor 的存活状态
```

3. 上次内容总结

第一次课主要讲解的是：

```
1、Flink RPC 详解
2、Flink 集群启动脚本分析
3、Flink 主节点 JobManager 启动分析
```

在 JobManager 启动过程中，主要做了四件大事：

```
1、初始化各种服务：initializeServices(....)
2、WebMonitorEndpoint 启动
3、ResourceManager 启动
4、Dispatcher 启动
```

第二次课主要讲解的是 Flink 集群启动的 从节点启动流程的源码剖析，主要涉及到的知识点：

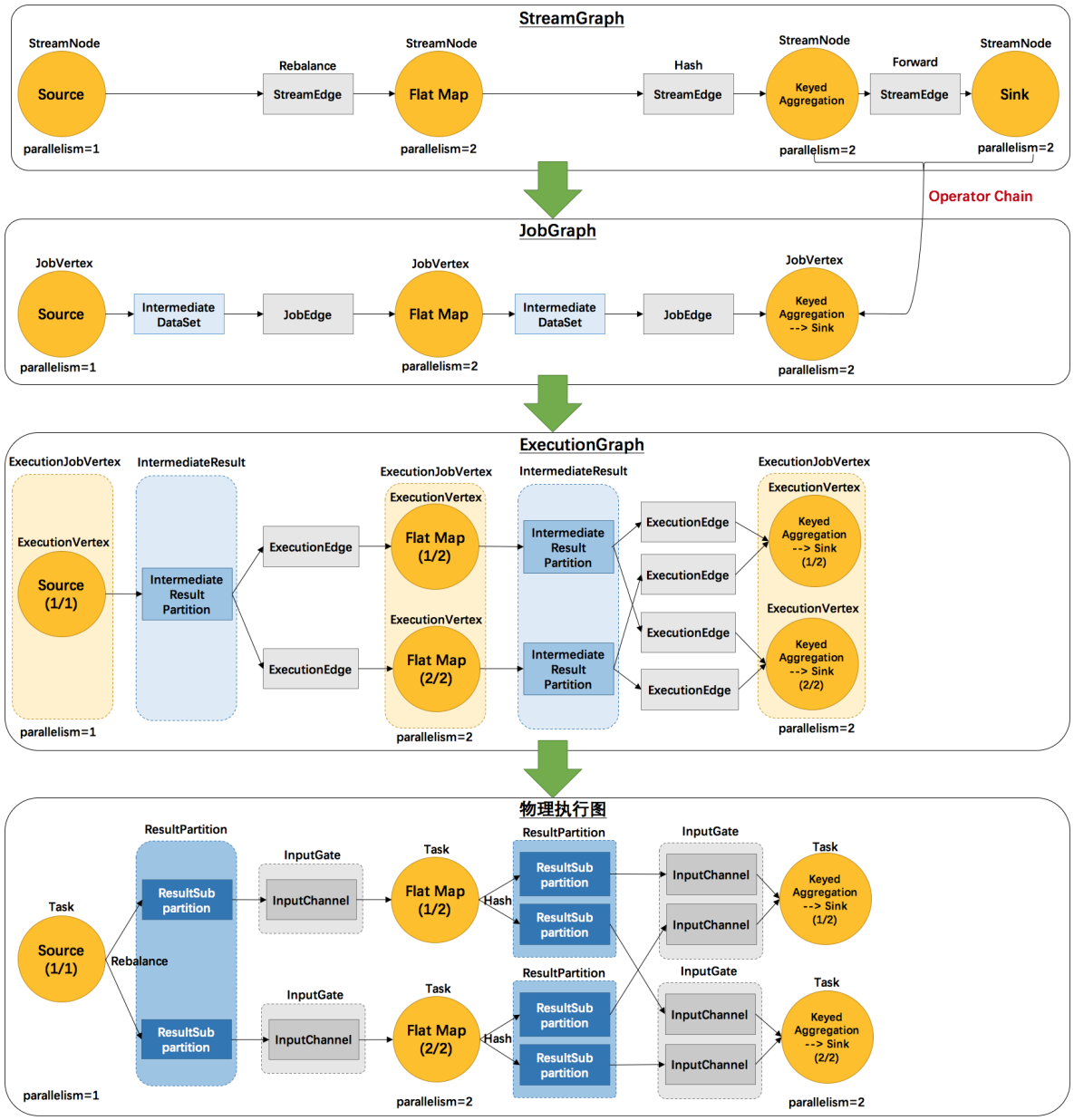
```
1、Flink TaskManager 启动源码分析
2、TaskManagerServices 初始化源码剖析
3、TaskExecutor 启动和源码分析
4、TaskExecutor 注册和心跳源码剖析
```

在这个 TaskManager 启动过程中，最重要的事情，就是在 TaskManager初始化了一些基础服务和一些对外提供服务的核心服务之后就启动 TaskExecutor，向 JobManager(ResourceManager) 进行 TaskManager 的注册，并且在注册成功之后，维持 TaskManager 和 JobManager(ResourceManager) 的心跳。

第三次课，也就是上次课程，主要讲解的是 Flink 的 Job 的提交：主要内容是 Flink 应用程序的提交。主要包含 Job 的三层 Graph 处理（首先根据应用程序构建 StreamGraph，然后构建 JobGraph，并行化之后生成 ExecutionGraph）。

```
1、Flink 编程套路总结
2、Flink 提交执行脚本分析
3、Flink CliFrontend 提交应用程序源码剖析
4、ExecutionEnvironment 源码解析
5、Job 提交流程源码分析
6、WebMonitorEndpoint 处理 RestClient 的 JobSubmit 请求
7、StreamGraph 构建和提交源码解析
8、JobGragh 构建和提交源码解析
9、ExecutionGragh 构建和提交源码解析
```

最重要的就是先理解 Flink 的四层 Graph 架构：



4. 本次内容大纲

今天的课程是 Flink 源码剖析 的第四次：在上一次讲解完 Flink 的 Job 是怎么提交和组织的之后，我们今天重点关注 Flink 的 Slot 管理和 Task 的具体执行细节！

```
1、JobMaster 启动执行
2、Slot 管理（申请）源码解析
3、Task 提交到 TaskExecutor 执行源码剖析
```

第五次源码课的主要内容大刚：就是在 Task 被部署运行了以后，要干的事情：

```
1、StreamTask 具体执行细节源码剖析
2、State 管理源码剖析
3、Flink Checkpoint 源码剖析
```

5. 详细课堂内容

5.1. JobMaster 启动执行

核心入口：

```
JobMaster.startJobExecution(newJobMasterId);
```

且看内部细节：

```
// 开始启动 JobMaster 来执行
JobMaster.startJobExecution(newJobMasterId);

    // 启动服务
    JobMaster.startJobMasterServices();
        // 启动 JobMaster 的心跳服务
        startHeartbeatServices();
            taskManagerHeartbeatManager =
heartbeatServices.createHeartbeatManagerSender(....);
            resourceManagerHeartbeatManager =
heartbeatServices.createHeartbeatManager(.....);
        // 开启 SotManagerImpl 服务
        slotPool.start(getFencingToken(), getAddress(),
getMainThreadExecutor());
            // 启动两个定时任务
            scheduleRunAsync(this::checkIdleSlot, idleSlotTimeout);
            scheduleRunAsync(this::checkBatchSlotTimeout, batchSlotTimeout);
        // 开启 Scheduler 服务
        scheduler.start(getMainThreadExecutor());
        // 联系 ResourceManager
        reconnectToResourceManager(new FlinkException("Starting JobMaster
component."));
            closeResourceManagerConnection(cause);
            tryConnectToResourceManager();
                resourceManagerConnection = new
ResourceManagerConnection(.....);
                resourceManagerConnection.start();
                    createNewRegistration();
                        generateRegistration()
                    newRegistration.startRegistration();
                        register(rpcGateway, 1, ....)
                            invokeRegistration(gateway, ....)
                                gateway.registerJobManager(....)
                                    // 内部实现，总共三件事
                                    // 第一件事：完成注册
                                    // 第二件事：维持 ResourceManager 和 JobMaster
之间的心跳
                                    // 第三件事：返回注册成功的消息：
JobMasterRegistrationSuccess
        // 监听 ResourceManager 地址
        resourceManagerLeaderRetriever.start(new
ResourceManagerLeaderListener());

    // 开始调度执行
    JobMaster.resetAndStartScheduler();
```

```
        JobMaster.startScheduling();
            // 由当前调用关系，可知，DefaultScheduler 是真正完成 JobMaster 的 Task 的
slot申请 和 部署运行的
            DefaultScheduler.startScheduling();
                // 第一件事： 开启 OperatorCoordinator
                startAllOperatorCoordinators();
                // 调度的内部实现
                startSchedulingInternal();
                    // 准备 ExecutionGraph 用于调度
                    prepareExecutionGraphForNgScheduling();
                        // 更改 Job 的状态为 RUNNING
                        executionGraph.transitionToRunning();
                    // 调度执行
                    schedulingStrategy.startScheduling();
                        // 申请 slot 和 部署 Task 执行
                        // 方法参数是获取到 调度拓扑中 的所有 ExecutionVertex ID

allocateSlotsAndDeploy(SchedulingStrategyUtils.getAllVertexIdsFromTopology(sched
ulingTopology));

schedulerOperations.allocateSlotsAndDeploy(executionVertexDeploymentOptions);
                                    // 申请 Slot
                                    allocateSlots(executionVertexDeploymentOptions);
                                    // 部署 Task
                                    waitForAllSlotsAndDeploy(deploymentHandles);
```

总结一下，大致做以下这些事情：

```
1、启动 JobMaster
    启动 JobMaster 运行过程中所需要的各种基础服务： slot管理，心跳服务，....
    JobMaster 去联系 ResourceManager 执行 Job 注册
2、Job 开始调度
    先解析 ExecutionGraph： 解析完毕之后就能知道到底会要启动多少Task，需要申请多少Slot
    申请 Slot
    执行 deploy 部署
```

5.2. Slot 管理（申请和释放）源码解析

核心入口：allocateSlots(executionVertexDeploymentOptions);

大体上，分为四个大步骤：

```
1、JobMaster 发送请求申请 slot
2、ResourceManager 接收到请求，执行 slot 请求处理
3、TaskManager 处理 ResourceManager 发送过来的 Slot 请求
4、JobMaster 接收到 TaskManager 发送过来的 Slot 申请处理结果
```

接下来看 TaskManager 的 slot 管理详细细节：

```
// JobMaster 发送请求申请 slot
DefaultScheduler.allocateSlots();
DefaultExecutionSlotAllocator.allocateSlotsFor();
NormalSlotProviderStrategy.allocateSlot();
```

```
SchedulerImpl.allocateSlot();
SchedulerImpl.allocateSlotInternal();
SchedulerImpl.internalAllocateSlot();
SchedulerImpl.allocateSingleSlot();
SchedulerImpl.requestNewAllocatedSlot();
SlotPoolImpl.requestNewAllocatedBatchSlot();
SlotPoolImpl.requestNewAllocatedSlotInternal();
SlotPoolImpl.requestSlotFromResourceManager();

// ResourceManager 接收到请求，执行 slot请求处理
ResourceManager.requestSlot();
SlotManagerImpl.registerSlotRequest();
SlotManagerImpl.internalRequestSlot();
SlotManagerImpl.allocateSlot();
TaskExecutorGateway.requestSlot();

// TaskManager 处理 ResourceManager 发送过来的 Slot 请求
TaskExecutor.requestSlot();
TaskExecutor.offerSlotsToJobManager();
TaskExecutor.internalOfferSlotsToJobManager();
JobMasterGateway.offerSlots();

// JobMaster 接收到 TaskManager 发送过来的 Slot 申请处理结果
JobMaster.offerSlots(...);
SlotPoolImpl.offerSlots(...);
SlotPoolImpl.offerSlots(...);
```

事实上，每个 reqeustSlot 请求其实申请到的就是一个 LogicalSlot

但是最后返回的结果是：`List<SlotExecutionVertexAssignment>`

5.3. Task 部署和提交

这个知识点，就是告诉你，一个 FLink Job 中的每一个 Task 到底都是怎么启动起来的，但是今天不涉及到 Task 的具体执行！

核心入口是：

```
waitForAllSlotsAndDeploy(deploymentHandles);
```

详细流程：

```
// DefaultScheduler 完成 Task 调度派发
DefaultScheduler.waitForAllSlotsAndDeploy();
DefaultScheduler.deployAll();
DefaultScheduler.deployOrHandleError();
DefaultScheduler.deployTaskSafe();
DefaultExecutionVertexOperations.deploy();
ExecutionVertex.deploy();
Execution.deploy();
TaskDeploymentDescriptorFactory.createDeploymentDescriptor();
RpcTaskManagerGateway.submitTask();
TaskExecutor.submitTask();

// TaskExecutor 接到 submitTask RPC 请求：TaskDeploymentDescriptor
```

```
TaskExecutor.submitTask();
    // 构造 Task 实例对象
    Task task = new Task(....)
    // 启动 Task 中的 executingThread 来执行 Task
    task.startTaskThread();
        Task.run()
            Task.doRun();
                // 先更改 Task 的状态： CREATED ==> DEPLOYING
                transitionState(ExecutionState.CREATED,
ExecutionState.DEPLOYING)
                final ExecutionConfig executionConfig =
serializedExecutionConfig.deserializeValue(userCodeClassLoader);
                // 初始化输入和输出组件
                setupPartitionsAndGates(consumableNotifyingPartitionWriters,
inputGates);
                // 注册 输出
                for(ResultPartitionWriter partitionWriter :
consumableNotifyingPartitionWriters) {

 taskEventDispatcher.registerPartition(partitionWriter.getPartitionId());
                }
                Environment env = new RuntimeEnvironment(....)
                // 初始化 调用对象
                // 两种最常见的类型： SourceStreamTask、OneInputStreamTask、
TwoInputStreamTask
                // 父类： StreamTask
                AbstractInvokable  invokable =
loadAndInstantiateInvokable(userCodeClassLoader, nameOfInvokableClass, env);

                // 先更改 Task 的状态： DEPLOYING ==> RUNNING
                transitionState(ExecutionState.DEPLOYING,
ExecutionState.RUNNING)

                // 真正把 Task启动起来了
                invokable.invoke();

                for(ResultPartitionWriter partitionWriter :
consumableNotifyingPartitionWriters) {
                    if(partitionWriter != null) {
                        partitionWriter.finish();
                    }
                }

                // 先更改 Task 的状态： RUNNING ==> FINISHED
                transitionState(ExecutionState.RUNNING, ExecutionState.FINISHED)
```

6. 本次课程总结

本次课程，主要讲解的是，Flink 客户端如何把一个 Job 提交到集群上去运行。主要涉及到三个知识点：

```
1、JobMaster 启动执行
2、Slot 管理（申请）源码解析
3、Task 提交到 TaskExecutor 执行源码剖析
```

下次课的主要内容，是 Task 执行构成中涉及到的一些细节，比如 State 管理，Checkpoint 等。

7. 本次课程作业

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

```
1、资源集群主节点叫做： ResourceManager，负责管理整个集群的资源
2、资源集群从节点叫做： TaskExecutor，负责提供资源
3、ResourceManager 启动的时候，要启动一个验活服务，制定一种机制（比如：某个 TaskExecutor 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
4、TaskExecutor 启动之后，需要向 ResourceManager 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 ResourceManager 之间的心跳以便 ResourceManager 识别到 TaskExecutor 的存活状态
```